Using Dynamic Sensitivity Analysis to Assess Testability

Jeffrey Voas*, Larry Morell†, Keith Miller‡

Abstract: This paper discusses *sensitivity analysis* and its relationship to random black box testing. Sensitivity analysis estimates the impact that a programming fault at a particular location would have on the program's input/output behavior. Locations that are relatively "insensitive" to faults can render random black box testing unlikely to uncover programming faults. Therefore, sensitivity analysis gives new insight when interpreting random black box testing results. Although sensitivity analysis is computationally intensive, it requires no oracle and no human intervention.

Index Terms: Sensitivity analysis, random black box testing, data state, data state error, testability, fault, failure.

*Supported by a National Research Council Resident Research Associateship.

†Supported by NASA Grant NAG-1-824.

‡Supported by NASA Grant NAG-1-884.

1 Introduction

Testing seeks to reveal software faults by executing a program and comparing the output expected to the output produced. Exhaustive testing is the only testing scheme that can (in some sense) guarantee correctness. All other testing schemes are based on the assumption that successful execution on some inputs implies (but does not guarantee) successful execution on other inputs. It is well known that some programming faults are very difficult to find using testing, and some work has been done on classifying types of faults [1]. But in this paper, we do not focus on the **faults** that exist; instead, we focus on **characteristics of a program** that will make faults (if they exist) hard to find using random black box testing. Thus correctness is not our direct concern. Instead, given a piece of code, we try to predict if random black box testing is likely to reveal any faults that exist in that code. The type of faults that we are considering in this paper are faults which have already been compiled into the code.

The *testability* of a program is a prediction of its ability to hide faults when the program is black box tested with inputs randomly selected from a particular input distribution. Testability is determined by the structure and semantics of the code and by an assumed input distribution. Thus, two different programs can compute the same function but may have different testabilities. A program is said to have *high testability* when it readily reveals faults through random black box testing; a program with *low testability* is unlikely to reveal faults through random black box testing. Low testability is a dangerous circumstance because considerable testing may succeed although the program has many faults.

A fault can lie anywhere within a program, so any method of determining testability must take into consideration all places in the code where a fault can occur. We use the term *location* to indicate a place in a program where a fault can occur. Although the techniques we propose can be used at different granularities, this paper concentrates on locations that roughly correspond to single commands in an imperative, procedural language.

We expect that any method for determining testability will require either extensive analysis, a large amount of computing resources, or both. However, the potential benefits for measuring testability are significant. If testability can be effectively estimated, we can gain considerable insight into several issues important to testing:

1. *Where to get the most benefit from limited testing resources:*

   A module with low testability requires more testing than a module with high testability. Testing resources can therefore be distributed more effectively.

2. *When to use some verification technique other than testing:*

   Extremely low testability suggests that an inordinate amount of testing may be required to gain confidence in the correctness of the software. Alternative techniques such as proofs of correctness or code review may be more appropriate for such modules.

3. *The degree to which testing must be performed in order to be convinced that a location is probably correct:*

   Testability can be employed to estimate how many tests are necessary to gain desired confidence in the correctness of the software.

4. *Whether or not the software should be rewritten:*

   Testability may be used as a guide to whether or not critical software has been sufficiently verified. If a piece of critical software has low testability, then it may be rejected because too much testing will be required to sufficiently verify a sufficient level of reliability.

In order to understand our approach to measuring testability, we define a new term, "sensitivity." We define the *sensitivity* of a location S to be a prediction of the probability that a fault in S will result in a software failure under some specified input distribution. If a location S has a sensitivity of .99 under a distribution D, then it is predicted that almost any input in the distribution that executes the location will cause a program failure. If a location has a sensitivity of .01 under D, then it is predicted that no matter what fault is present at S, few inputs that execute S would cause the program to fail.

Sensitivity is clearly related to testability, but the terms are not equivalent. Sensitivity focuses on a single location within a program and the impact a fault at that location can have on the input/output behavior of the program; testability encompasses the whole program and its sensitivities under a given input distribution. In this paper we discuss a means whereby the testability of the whole program can be estimated from the sensitivities of its locations. *Sensitivity analysis* is the process of determining the sensitivity of a location in a program. From the collection of sensitivities over all locations, we determine the testability of the program.

One method of performing sensitivity analysis is discussed here, called PIE analysis (for Propagation, Infection, and Execution analysis). PIE is dynamic in the sense that it requires execution of the code. Inputs are randomly selected from the input distribution and the computational behavior of the code on these inputs is compared against the behavior of similar code (similar in ways described later). PIE analysis is dynamic, but it is not software testing because no outputs are checked against a specification or oracle. The next section of this paper describes the computational behavior we are investigating in a manner similar to [3]. Next we describe the PIE method of sensitivity analysis. We then show how to use PIE results to quantify the testability of a program.

2 Fault/Failure Model

If the presence of faults in programs guaranteed program failure, every program would be highly testable. To understand why this is not the case, it is necessary to consider the sequence of location executions that a program performs. In this discussion, a *computation* refers to an execution trace in which the value of each variable is displayed after the execution of each location for some input. A particular trace is produced by a program in response to a particular input. Each set of variable values displayed after the execution of a location in a computation is called a *data state.* After executing a fault the resulting data state might be corrupted; if there is corruption in a data state, we say that *infection* has occurred and the data state contains an *error*, termed a *data state error.*

The program, P, in Figure 1 is intended to display, if possible, an integral solution to the quadratic equation $ax^2 + bx + c$, for integral a, b, and c (we

```
{1} read (a,b,c);
{2} if a <> 0 then begin
{3}    d := b*b − 5*a*c;
{4}    if d < 0 then
{5}       x := 0
       else
{6}       x := (−b + trunc(sqrt(d))) div (2*a)
    end
    else
{7}    x := − c div b;
{8} if (a*x*x + b*x + c = 0) then
{9}    writeln(x, ' is an integral solution')
    else
{10}   writeln('There is no integral solution')
```

Figure 1. Program P.

have fixed a, b, and c such that a and c are in [0,10] and b is in [1,1000]). The program has a fault at line 3: the constant 5 should be the constant 4. Each computation of P falls into one of four categories: (1) the fault is not executed; (2) the fault is executed, but does not infect any data state; (3) the fault is executed, some data state or states are infected, but the output is correct anyway; or (4) the fault is executed, infection occurs, and the infection causes an incorrect output. Only computations in the final category would make the fault *visible* to a tester. Here are examples of each type of computation of P:

1. The computation for the input $(a, b, c) = (0, 3, 6)$ is displayed in Figure 2. The value of $a = 0$ causes the selection of a path that does not include the fault. Clearly any such execution will not fail.

2. The computation for input (3,2,0) is shown in Figure 3. The fault is

Location	a	b	c	d	x	Output
1	0	3	6	undefined	undefined	
7	0	3	6	undefined	-2	
8	0	3	6	undefined	-2	
9	0	3	6	undefined	-2	-2 is an integral solution

Figure 2. (0,3,6) as input to P.

Location	a	b	c	d	x	Output
1	3	2	0	undefined	undefined	
2	3	2	0	undefined	undefined	
3	3	2	0	4	undefined	
6	3	2	0	4	0	
8	3	2	0	4	0	
9	3	2	0	4	0	0 is an integral solution

Figure 3. (3,2,0) as input to P.

reached but the computation proceeds just as if there were no fault present, because $c = 0$ prevents the fault from impacting the computation. No infection has occurred.

3. For input (1,−1,−12) the fault infects the succeeding data state, producing $d = 61$ instead of $d = 49$ (see Figure 4). This data state error then *propagates* to location 6 where it is *canceled* by the integer square root calculation, because 7 is computed in either case.

4. Executing the program with input (10,0,10), executes the fault which then infects the succeeding data state in such a way that the data state error propagates to the output (See Figure 5).

Computation type (1) demonstrates that an execution of P can only reveal information about the portion of the code that is executed. Computation

Location	a	b	c	d	x	Output
1	1	-1	-12	undefined	undefined	
2	1	-1	-12	undefined	undefined	
3	1	-1	-12	61	undefined	
6	1	-1	-12	61	4	
8	1	-1	-12	61	4	
9	1	-1	-12	61	4	4 is an integral solution

Figure 4. $(1, -1, -12)$ as input to P.

Location	a	b	c	d	x	Output
1	10	0	10	undefined	undefined	
2	10	0	10	undefined	undefined	
3	10	0	10	-500	undefined	
4	10	0	10	-500	undefined	
5	10	0	10	-500	0	
10	10	0	10	-500	0	There is no integral solution

Figure 5. $(10, 0, 10)$ as input to P.

types (2) and (3) provide a false sense of security to a tester because the fault is executed but no visible failure results. There are two reasons that an executed fault may remain hidden during random black box testing: a lack of infection (meaning a data state error is not created) and a lack of propagation (meaning a data state error ceases to exist to the output). Computation type (4) illustrates three necessary and sufficient conditions for a fault to produce a failure:

1. The fault must be *executed.*

2. The succeeding data state must be *infected.*

3. The data state error must *propagate* to output.

These three phenomena comprise the fault/failure model. This model underlies the dynamic method discussed here to determine the sensitivity of a given location in the code. Complete sensitivity analysis requires every location to be analyzed for three properties: the probability of execution occurring, the probability of infection occurring, and the probability of propagation occurring. An execution-based method for estimating these probabilities is discussed in the following section.

3 Sensitivity Analysis: Execution, Infection, and Propagation

The previous section introduced the three part model of software failure: (1) execution, (2) infection, and (3) propagation to output. In this section we use this model to analyze the sensitivity of locations to potential (and unknown) faults. Sensitivity analysis is decomposed into execution analysis, infection analysis, and propagation analysis—one type of analysis to handle each part of the model.

3.1 Obtaining Three Estimates

All three analyses can be accomplished at several different levels of abstraction: programs, modules, and statements are three such levels. In this section we describe an analysis done on program locations where a location is a unit of code that either changes the value of a variable, changes the flow of control, or produces an output. A program location is similar to a single high level statement, but some such statements contain multiple locations. (For example, **read**(a, b) contains two locations.)

3.1.1 Execution Analysis

Execution analysis is the most straightforward of the three analyses. *Execution analysis* requires a specified input distribution (as does any quantifiable testing method). Execution analysis executes the code with random inputs from that distribution and records the locations executed by each input. This produces an estimate of the probability that a location will be executed by a randomly selected input according to this distribution. The estimate of this probability is termed an *execution estimate*. Hence execution analysis is concerned with the likelihood that a particular location will have an opportunity to affect the output.

The algorithm for finding an execution estimate is:

1. Set array `count` to zeroes, where the size of `count` is the number of locations in the program being analyzed,

2. Instrument the program with "write" statements at each location that print the location number when the location is executed, making sure that if a location is repeated more than once on some input, the "write" statement for that location is only executed once for that input,

3. Execute n input points on the "instrumented" program, producing n strings of location numbers,

4. For each location number l in a string, increment the corresponding `count[`l`]`. If a location k is executed on every input, `count[`k`]` will equal n,

5. Divide each element of `count[l]` by n yielding an execution estimate for location l.

Each execution estimate is a function of the program and an input distribution.

3.1.2 Infection Analysis

Infection analysis estimates the probability that a mutant at a particular location will adversely affect the data state which immediately results when the location is executed. In other words, will the mutant produce a value in the following data state that is different than the value that is produced by the original location?

Infection analysis is similar to mutation testing; what is different is the information collected [4]. For a given location in a program, we do not know whether or not a fault is present, and we don't know what types of faults are possible at the location. So we create a set of mutants at each location. After creating a set of mutants, we obtain an estimate of the probability that the data state is affected by the presence of a mutant for each mutant in the set. We select a mutant from the set, mutate the code at the location, and execute each resulting mutant many times. The data states created by executing the mutants are checked against the data states from the original location to determine if the mutants have *infected* the data states. The proportion of executions that infect for a particular mutant are the *infection estimate* for that mutant. In the next section we describe how we extract a single infection estimate from the set of infection estimates for the location.

The algorithm for finding an infection estimate is:

1. Set variable `count` to 0,
2. Create a mutant for location l denoted as h,
3. Present the original location l and the mutant h with a randomly selected data state from the set of data states that occur immediately prior to location l and execute both locations in parallel,
4. Compare the resulting data states and increment `count` when the function computed by h does not equal the function computed by l for this particular data state,
5. Repeat algorithm steps 3 and 4 n times,
6. Divide `count` by n yielding an infection estimate.

An infection estimate is a function of a location, the mutant created, and the set of data states which occur before the location. This algorithm is generally performed many times at a location to produce a set of infection estimates.

The exact nature of the best code mutations for infection analysis is still being researched, but we have obtained encouraging results from a small set of mutations based on semantic changes. These mutations are straightforward to describe, and can be automated. Furthermore, the results of the analysis using these mutations has been encouraging. To illustrate this set of mutants, Figure 8 shows the mutants generated for locations 3, 4, and 6 of program P. For each mutant, we give the infection estimate obtained from executing 10000 random inputs through each.

3.1.3 Propagation Analysis

Propagation analysis estimates the probability that an infected data state at a location will propagate to the output. To make this estimate, we repeatedly *perturb* the data state which occurs after some location of a program, changing one value in the data state (hence one *live* variable receives an altered value) for each execution. We term a variable as being live at a particular location of a program if the variable has any potential of affecting the output of the program. For instance, a variable which is defined but never referenced is one example of a variable which would not be live. By examining how often a forced change into a data state affects the output, we calculate a *propagation estimate*, which is an estimate of the affect that a live variable (the variable that received the forced change into the data state) has on the program's output at this location. We find a propagation estimate for a set of live variables at each location (assuming there is more than one live variable at a location), thus producing a set of propagation estimates—one propagation estimate per live variable.

If we were to find that at a particular location a particular live variable had a near 0.0 propagation estimate, we would realize that this variable had virtually no affect on the output of the program at this location. This does not necessarily mean that this variable has no affect on the output—only that it has little effect. Hence propagation analysis is concerned with the likelihood that a particular live variable at a particular location will cause the output to differ after the live variable's value is changed in the location's data state. In the next section we describe how we extract a single propagation estimate from the set of propagation estimates for the location.

Propagation analysis is based on changes to the data state. To obtain the data states that are then executed to completion, we use a mathematical function based upon a random distribution termed a *perturbation function.* A perturbation function inputs a variable's value and produces a different value chosen according to the random distribution—the random distribution uses the original value as a parameter to in when producing the different value. We are researching different perturbation functions—we currently use a uniform distribution whose mean is the original value. The maximum and minimum "different" values which can be produced by the perturbation function are determined by the range of values that a variable had during the executions used to obtain the execution estimates.

An algorithm for finding a propagation estimate is:

1. Set variable `count` to 0,

2. Randomly select a data state from the distribution of data state that occur after location l,

3. Perturb the sampled value of variable a in this data state if a is defined, else assign a a random value, and execute the succeeding code on both the perturbed and original data states,

4. For each different outcome in the output between the perturbed data state and the original data state, increment `count`; increment `count` if an infinite loop occurs (set a time limit for termination, and if execution is not finished in that time interval, assume an infinite loop occurred),

5. Repeat algorithm steps 2-4 n times,

6. Divide `count` by n yielding a propagation estimate.

A propagation estimate is a function of a location, a live variable, the set of data states which occur after the location (which are a function of some input distribution), and the code which is possibly executed after the location.

Execution, infection, and propagation analyses each involve significant execution time, since bookkeeping, mutating, and perturbing are done on a location by location basis. However (unlike testing) none of this analysis requires an oracle to determine a “correct” output. Instead, we can detect *changes* from the original input/output behavior without determining correctness. This allows the entire sensitivity analysis to be automated. Sensitivity analysis, though computationally intensive, is not labor intensive. This emphasis on computing power seems increasingly appropriate as machine executions become cheaper and programming errors become more expensive.

3.2 Understanding the Resulting Estimates

When all three analyses are complete, we have three sets of probability estimates for each location:

1. Set 1: The estimate of the probability that a location is executed (one number);

2. Set 2: The estimates of the probabilities, one estimate for each mutant at the location, that given the location is executed, the mutant will adversely affect the data state; and

3. Set 3: The estimates of the probabilities, one estimate for each live variable at the location, that given that the variable in the data state following the location is infected, the output will be changed.

For each location there are several ways to manipulate these three estimates, but we will discuss only one. Others may be found in [2]. In order to reveal a fault at a particular location with a particular test case, the location must be executed, an infection must occur, and the infection must propagate to the output. If any of these does not occur, the fault will be "invisible" to the tester. Therefore, a conservative estimate of the sensitivity for a location can be derived by using the minimum estimate from each of these three sets.

We choose the minimum estimate from each set because we think it is better to overestimate the amount of random black box testing needed to reveal a fault, rather than to underestimate it. If we underestimate the amount of testing necessary for a program, we may be fooled into thinking no fault is there when there really is a fault there. If we overestimate the amount of testing required, the worst that can happen is that we waste testing resources by performing too much testing. We prefer the latter outcome, and therefore adopt the conservative approach.

The next section describes how the numbers obtained from our sensitivity analysis correlated with actual testing behavior in an experiment.

4 Relating Sensitivity to Testability

We produce the sensitivity of a location from the propagation, infection, and execution estimates at that location. Recall that in the fault/failure model, there are three conditions necessary for a software failure. The *failure probability* of a program is the probability that some randomly selected input according to some distribution will produce a failure. For a correct program which is injected with a specific fault at a single location, the failure probability of the program is the product of the probability of execution, infection, and propagation occurring for this fault. In the "real-world" scenario that we are in, we do not have the luxury of knowing about the occurrence of a specific fault at a specific location. What we have available is the current program, without a specification or oracle, and an input distribution. Thus we use the estimates obtained from the algorithms to estimate the affect that a fault at this location, *if a fault exists*, would have on the output.

A simple method for finding the sensitivity of a location l is to multiply the minimum propagation estimate, minimum infection estimate, and execution estimate of a location to get the location's sensitivity. As described above, we have chosen to select the minimum infection and propagation estimates available at a location in attempts to avoid overestimating a location's sensitivity.

Each estimate produced by our algorithms has an associated confidence interval; we choose to take the lower bound on the interval as reflected in equation 1. Equation 1 is our first attempt at finding the sensitivity of some location l.[1]

$$\varepsilon_l \cdot (\min[\psi_{l,j}])_{min} \cdot (\min[I_{l,a}])_{min} \tag{1}$$

[1] $(\cdot)_{min}$ denotes the lower bound for the confidence interval for an estimate and $(\cdot)_{max}$ denotes the upper bound for the confidence interval for an estimate. ε_l represents the execution estimate for location l. $\psi_{l,j}$ represents the propagation estimate at location l after live variable j is perturbed. $I_{l,a}$ represents the infection estimate created by mutant a at location l.

But propagation estimates are a function of the infections created by random distributions, not infections created by specific mutants. In the worst case possible, the proportion of data state errors created by the mutant that produces the minimum infection estimate is *exactly* the proportion of data state errors that did not propagate when the minimum propagation estimate was produced. Although unlikely, since we want an underestimated sensitivity instead of an overestimated sensitivity, we modify equation 1 to account for this possibility and produce the sensitivity of a location l (denoted as β_l):

$$\beta_l = \varepsilon_l \cdot [\sigma((\min[I_{l,a}])_{min}, (\min[\psi_{l,j}])_{min})] \tag{2}$$

where

$$\sigma(a,b) = \begin{cases} a-(1-b) & \text{if } a-(1-b)>0 \\ 0 & \text{otherwise.} \end{cases}$$

In order to explore the relationship between sensitivity analysis and testability, we performed an experiment. PIE was performed at locations 3, 4, and 6 of program P with 10,000 random inputs and the following random input distributions: a and c were equilikely[0,10], and b was equilikely[1,1000]. Hence there are 121000 distinct inputs to this program. The results of execution analysis appear in Figure 7, the results of propagation analysis appear in Figure 6, and the results of infection analysis appear in Figure 8.

The sensitivities found using equation 2 for the locations 3, 4, and 6 are all 0.0. Locations 3 and 6 produced very low propagation estimates, and location 4 produced a very low infection estimate, thus resulting in zero sensitivities. At first glance, this seems to be less than informative, however this information is both reasonable and useful.

Location 8 is critical in reducing the propagation estimates found from the

Location	Perturbed variable	Propagation estimate
3	d	0.076
3	a	0.0015
3	b	0.0822
4	d	0.076
4	a	0.0015
4	b	0.0822
6	b	0.00044
6	x	0.09
6	a	0.00033

Figure 6. Propagation estimates for locations 3, 4, and 6 of P.

Location	Execution estimate
3	0.9083
4	0.9083
6	0.901

Figure 7. Execution estimates for locations 3, 4, and 6 of P.

preceding 7 locations. Locations 2-7 in P do virtually all of the computing of x in P. At location 8, unless its condition is true, computation of x in locations 2-7 is not referenced in the output; the program prints out that there is no solution, and whatever computations occurred in locations 2-7 are ignored. This is a program that by it's nature rarely has a solution with the particular input distribution we selected. (We purposely selected this input distribution to highlight what we mean by low sensitivities.) Hence, data state errors that are injected into locations 3, 5, 6, or 7 rarely are given the opportunity to affect the output, because location 9 has a low execution estimate. The low sensitivities for these locations reflect reality—with the input distribution given, data state errors in locations 3-7 will have little if

Location	Mutant	Infection estimate
3	d := **sqr**(b) − 4 ∗ a ∗ c;	0.912
3	d := **sqr**(b) − 5 ∗ c ∗ c;	0.818
3	d := **sqr**(c) − 5 ∗ a ∗ c;	0.999
3	d := **sqr**(b) − 5 ∗ a ∗ a;	0.906
3	d := **sqr**(b) + 5 ∗ a ∗ c;	0.912
3	d := **sqr**(b) ∗ 5 ∗ a ∗ c;	1.0
4	**if** (d<=0) **then**	0.00022
4	**if** (d<10) **then**	0.0011
4	**if** (d=0) **then**	0.01
6	x := (−a + **trunc**(**sqrt**(d))) **div** (2∗a)	0.99
6	x := (−b + **trunc**(**sqr**(d))) **div** (2∗a)	0.99
6	x := (−b + **trunc**(**sqrt**(d))) **div** (2)	0.059

Figure 8. Infection estimates for locations 3, 4, and 6 of P.

no affect on the output. Thus, under this distribution, P has low testability.

We now shift our attention to program M in Figure 10. Location 2 in M has a higher sensitivity than the locations in P. Since location 2 is an output location, the propagation estimate is 1.0, since any change to the output data state changes the output. Since the execution estimate at this location is 1.0, the sensitivity of location 2 in M depends almost entirely on the minimum infection estimate of location 2. Figure 11 shows infection estimates found for a small set of mutants tried at location 2. As can be seen, the minimal infection estimate mutant from this set of 0.8237, which is also the sensitivity (since $1 \cdot (0.8237 - (1 - 1)) = 0.8237$). Hence the minimum over the set of infection estimates becomes the sensitivity of location 2. For this location, sensitivity analysis predicts that this location will not require much black box testing to reveal a fault in location 2 if one exists. M is a program with high testability.

A "blind" experiment using P was run to test the hypothesis that sensitivity analysis is helpful in estimating testability. The hypothesis of this blind experiment among the authors was that *for an injected fault, the sensitivity for the location where the fault was injected was always less than or equal to the resulting failure probability estimate of any fault injected at that location.* Notice that we use sensitivities (which are found solely from P with a fault) to underestimate the failure probabilities that occur from a set of injected faults into the oracle version of P. One author produced the sensitivities already shown, while another author independently produced failure probability estimates after placing faults at locations 3, 4, and 6 in P. The failure probability estimates are based on 10000 inputs for the faults injected at locations 3 and 6, and 100000 inputs for the fault injected at location 4. The resulting failure probability estimates and faults are in Figure 9. In each case, the hypothesis was supported.

The sensitivity of a location is in some sense an indicator of how much testing is necessary to reveal a fault at that location. For instance, a sensitivity of 1.0 at some location suggests that on the first test of the location, failure will result if there exists a fault in the location, and hence the existence of a fault will immediately be revealed. A sensitivity of 0.01 suggests that, on average, 1 in every 100 tests of a location will reveal a failure if a fault exists. Sensitivity gives a rough estimate of how frequently a fault will be revealed if one exists at a location.

The sensitivity β_l can be used as an estimate of the minimum failure probability for location l in the equation $1 - (1 - \beta_l)^T = c$, where c is the confidence that the actual failure probability of location l is less than β_l. With this equation, we can obtain the number of tests T needed for a particular

Location	Injected fault	Failure probability estimate
3	d := **sqr**(a) − 5 ∗ a ∗ c;	0.8970
4	**if** (d<=0) **then**	0.00012
6	x := (−b + **trunc**(**sqrt**(d))) **div** 2 ∗ a	0.9001

Figure 9. Failure probability estimates and mutants of P.

```
{1} read (a,b,c);
{2} writeln (a*a + b*a + c);
```

Figure 10. Program M.

c. To obtain a confidence c that the true failure probability of a location l is less than β_l given the sensitivity of the location, we need to conduct T tests, where

$$T = \frac{\ln(1-c)}{\ln(1-\beta_l).} \tag{3}$$

When $\beta_l \approx 0.0$, we effectively have the confidence c after T tests that location l does not contain a fault. To obtain a confidence c that the true failure probability of a program is less than β_l given the sensitivities of its locations, we need to conduct T tests, where

$$T = \frac{\ln(1-c)}{\ln(1-\min_l[\beta_l]).} \tag{4}$$

When $\min_l[\beta_l] \approx 0.0$, we effectively have the confidence c after T tests that the program does not contain a fault. Note that for equation 3 and equation 4, β_l can not be zero or one.

We conservatively estimate the testability of an entire program to be the

Location	Mutant	Infection estimate
2	**writeln** (a*a + b*a + a)	0.9085
2	**writeln** (a*a + b*a -c)	0.9135
2	**writeln** (a*a - b*a +c)	0.909
2	**writeln** (a*c + b*a +c)	0.8237

Figure 11. Infection estimates for location 2 of M.

minimum sensitivity over all locations in the program:

$$\min_{l}[\beta_l] \tag{5}$$

Hence the greater the testability of a program, the fewer tests T needed in equation 4 to achieve a particular confidence c that the program does not contain a fault.

5 Conclusions

Sensitivity analysis is a dynamic method for analyzing code. This analysis embodies a three part fault/failure model: execution, infection, and propagation. The method we describe for estimating sensitivity is computationally intensive, but does not require an oracle or human intervention. In several experiments, a conservative interpretation of sensitivity analyses successfully identified locations that could easily hide faults from random black box testing.

Sensitivity analysis can add another dimension to software quality assurance. During initial code development, sensitivity analysis can identify code that will inherit a greater tendency to hide faults during testing; such code can be rewritten or can be subjected to other types of analysis to detect faults. During random black box testing, sensitivity analysis will help interpret testing results; we will have more confidence in code with high sensitivity that reveals no errors during random black box testing than code with low sensitivity that reveals no errors during the same type of testing. Testing resources can be saved by testing high sensitivity locations less frequently than might otherwise be necessary to obtain confidence in the code; conversely, low sensitivity locations may require additional testing. During maintenance, sensitivity analysis can be used to identify locations where subtle bugs could be hiding from conventional testing techniques; alternative analyses could then be used at those locations.

Sensitivity analysis includes some characteristics that are similar to mutation testing, but the differences are significant. Infection analysis mutates source code, and as such is related to mutation testing; however, the goals of the two techniques are distinct. Mutation testing seeks an improved set of test data; infection analysis seeks to identify locations where faults are unlikely to change the data state. Propagation analysis mutates the data state, not the code, and then examines whether or not the output is effected. This is similar to some data flow research, but again the aim is distinct. Sensitivity analysis dynamically estimates relevant probabilities, and uses these estimates to better understand test results; to our knowledge, this emphasis is unique.

6 References

[1] William E. Howden. "A Functional Approach to Program Testing and Analysis." *IEEE Transactions on Software Engineering*, October 1986.

[2] Jeffrey M. Voas and Larry J. Morell. "Applying Sensitivity Analysis Estimates to a Minimum Failure Probability for Software Testing." *Proceedings of the Eighth Pacific Northwest Software Quality Conf.*, October 1990.

[3] L. J. Morell. "A Theory of Fault-Based Testing." *IEEE Transactions on Software Engineering*, August 1990.

[4] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer*, pages 34–41, April 1978.